

*FINAL
11-1-96
OCT.
1996*

**FINAL PROGRESS REPORT
October 1996**

NASA GRANT NUMBER NCC 2-800

**Rapid Assessment of Agility for Conceptual
Design Synthesis**

by

**Dr. Daniel J. Biezad
Principal Investigator
Cal Poly State University
(805) 756-5126**

Presented to

**NASA Technical Monitor:
Mr. Paul Gelhausen
NASA Ames Research Center
Moffett Field, CA. 94035-1000**

October 1996

TABLE OF CONTENTS

LIST OF FIGURES	iv
Abstract	1
Introduction	2
Simulated Airport	2
Landing Task	3
HUD Symbolology	4
Up-and-Away Task	5
Computer Code	6

LIST OF FIGURES

Figure	Page
1 - Out the Window Scene Showing the Simulated Airport and HUD Symbology Overlay	3
2 - Target for Up-and-Away Task with HUD Overlay	4

Abstract

This project consists of designing and implementing a real-time graphical interface for a workstation-based flight simulator. It is capable of creating a three-dimensional out-the-window scene of the aircraft's flying environment, with extensive information about the aircraft's state displayed in the form of a heads-up-display (HUD) overlay. The code, written in the C programming language, makes calls to Silicon Graphics' Graphics Library (GL) to draw the graphics primitives. Included in this report is a detailed description of the capabilities of the code, including graphical examples, as well as a printout of the code itself.

Introduction

In order for the Aeronautical engineering student to competently analyze and design aircraft, he must be familiar with the way aircraft fly. In other words, he must have an intuitive, as well as mathematic, understanding of aircraft performance and, to some degree, aircraft handling qualities. A powerful tool for this is the flight simulator. An engineer can choose a specific set of flight conditions and maneuvers, or pilot tasks, and with the corresponding set of stability derivatives, analyze how well the aircraft flies.

Many flight simulators work by mathematically analyzing the flight conditions and stability derivatives, then giving a time history of important variables as output. While this information is useful, it often fails to provide the engineer an intuitive feel for what the aircraft is actually doing. By providing a graphical interface, such as an animated three-dimensional model of the world outside the aircraft, the engineer can actually see the dynamic responses. He can easily vary his inputs and rapidly and intuitively understand the effects of the changes.

Another important learning tool is the incorporation of real-time input into such a simulator, as opposed to a batch mode operation. This feature would allow the engineer to be in the loop, to become the pilot. In addition to flying the aircraft at the given flight conditions, the engineer would have the ability, through the mouse, keyboard, or flight stick, of actually changing the control inputs and immediately see the effects of his changes. In this way, the engineer can rapidly assess both qualitatively and quantitatively how well the aircraft performs.

This project consists of designing and implementing a real-time graphical interface for a workstation-based flight simulator. It is capable of creating a three-dimensional out-the-window scene of the aircraft's flying environment, with extensive information about the aircraft's state displayed in the form of a heads-up-display (HUD) overlay. The code, written in the C programming language, makes calls to Silicon Graphics' Graphics Library (GL) to draw the graphics primitives. Included in this report is a detailed description of the capabilities of the code, including graphical examples, as well as a printout of the code itself.

Simulated Airport

As shown in Figure 1, the out-the-window scene consists of a ground plane with a grid for visual reference, an airport, and some simple mountains in the distance. The airport is modeled after Moffet Naval Air Station, where NASA-Ames is located, and contains its two parallel runways, the blimp hangar, and two P-3 Orion submarine-hunter hangars. To give a sense of scale, the blimp hangar is 300 ft wide and 1,200 ft long. The runways at Moffet are built along a magnetic course of 320/140. Viewed from the south, the runway on the left (32L) is 8,125 ft long and 200 ft wide. The runway on the right is 9,200 ft long and 200 ft wide.

For simplicity in setting up the landing approach task in the simulator, some artistic license was taken, and the runways are drawn to align with magnetic north. For this reason, runways 32L and 32R become 36L and 36R, respectively. Therefore, when lined up with the runway in an approach from the south, the heading indicator in the simulator will read a bearing of 360. To improve the update rate of the graphics, markings are added only to the left runway (36L), and the right one left blank. The markings are constructed according to the standards for a precision-approach equipped runway, with some simulated tire marks for visual effect.

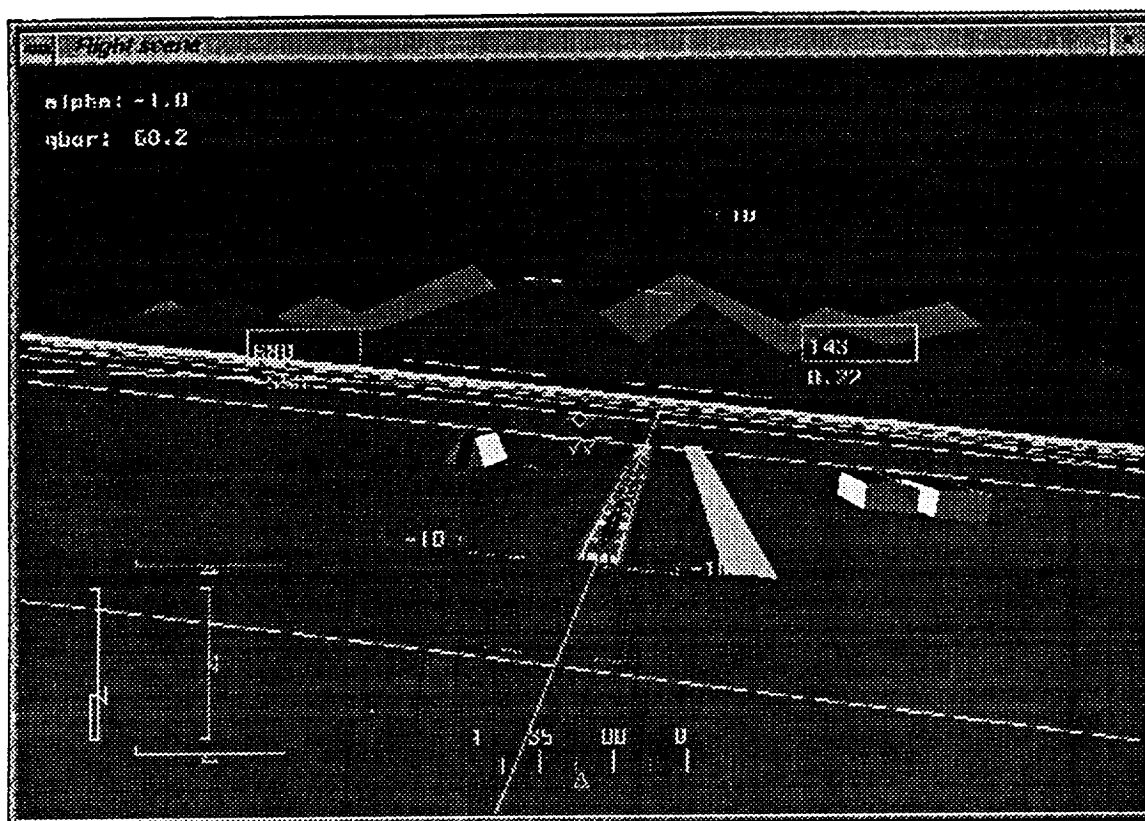


Figure 1 - Out the Window Scene Showing the Simulated Airport and HUD Symbology Overlay

Landing Task

To aid in the evaluation of a simulated aircraft's handling qualities, an approach to landing is one of the available tasks for the simulator pilot. A score is determined by integrating the squares of the distances above or below glideslope, and left or right of the runway centerline. To accomplish the landing task accurately, the pilot must be provided with an indication of position relative to the desired glideslope. Off of the approach end of runway 36L is a set of "telephone poles" that create a visual reference, as shown in Figure 1. The poles are arranged in two rows, angled outward from the touch-down point to create a tapered hallway that narrows as the aircraft approaches the runway. The poles get progressively taller away from the touch-down point, defining a plane that is the standard 3° glide slope. The approach can be set up steeper, to simulate an aircraft-carrier approach for example, by changing the glide slope angle in the input file that controls the simulation (*sim_ctrl.dat*). When the pilot is approaching the runway on the defined glide slope, the tops of the poles will line up parallel to the horizon. If the pilot is below glideslope, the tops of the poles will appear to angle upward. If flying above the glideslope, they appear to angle downward. With a little familiarization, the visual cues are quite sensitive to vertical deviations, especially closer to touch down. The two rows also create an additional visual cue that assists in lining up with the runway from a distance.

HUD Symbolology

On top of the out-the-window display is a collection of symbology that simulates a heads-up-display (HUD). As shown in Figure 1 and Figure 2, in the top left corner of the screen are digital indications of angle of attack (α) and dynamic pressure (q_{bar}), the two independent variables that define the table look up for stability derivatives in the simulation. When the aircraft flies outside the defined envelope of data, these values turn red, indicating to the pilot that the current flight state is not being properly modeled. The aircraft altitude is displayed in feet in a box on the left side of the screen, with the rate of climb in feet per minute under it. In the box on the right side is the true airspeed in knots (nautical miles per hour), with the Mach number shown below.

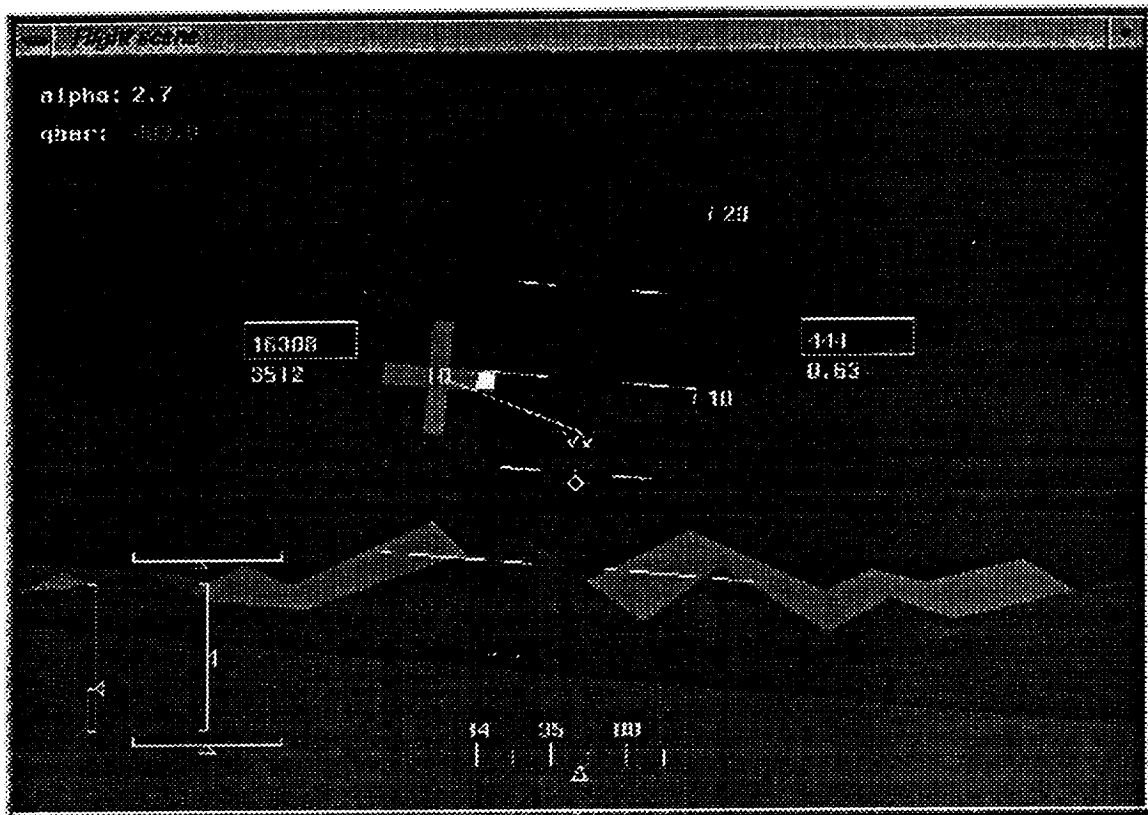


Figure 2 - Target for Up-and-Away Task with HUD Overlay

The center of the screen is dominated by the pitch ladder, which indicates the aircraft's pitch and bank angles. The "W" shaped symbol is the "waterline," and it represents the direction the nose of the airplane is pointed. In other words, the rung of the pitch ladder that it lies on corresponds to the Euler angle θ , or the angle between the aircraft body axis out the nose and the horizontal plane. The small diamond with three short lines coming out of it is the velocity vector, and it shows the actual direction of flight. The vertical displacement between this symbol and the waterline is therefore the angle of attack α , and the horizontal displacement indicates the angle of sideslip β . Since the rungs of the pitch ladder remain aligned with the

horizon, the aircraft bank angle ϕ is reflected in the ladder's rotation. At the bottom center of the screen is a heading indicator, or compass, that indicates the Euler angle Ψ .

In the bottom left corner of the window is a set of symbols that display the commanded thrust, actual thrust, and the positions of the ailerons, elevator and rudder. The vertical slider farthest to the left is the thrust indicator. The small triangle moves up and down in response to the throttle inputs from the pilot, from 0% at the bottom to 100% at the top. The rectangle on the left side of the line represents the actual thrust, and may lag behind the pilot's commands if the simulated engine is modeled to take time as it spools up and down. The other vertical slider shows the actual position of the elevator, which may be different from the commanded position if a feedback control law is implemented. For a conventional stable aircraft, full elevator deflection trailing edge down (stick forward, nose down pitching moment) drives the pointer to the top of the line, and vice-versa. The horizontal slider at the top of this cluster indicates the actual differential aileron deflection, with a full left-rolling deflection driving the triangle to the left end of the line, right-rolling to the right end. Similarly, the slider at the bottom of the cluster indicates actual rudder deflection. Rudder trailing edge full left (nose left yawing moment) drives the slider to the left end of the line, full right deflection to the right end.

Up-and-Away Task

The "up-and-away" task, which simulates an air-to-air engagement, is shown in Figure 2. The target is a cruciform shape, with a yellow light on each arm. The target moves relatively slowly, creating a gross acquisition/low frequency tracking task for the simulator pilot. If the target is lost from the field of view, the line drawn from the middle of the HUD to the center of the target will indicate the direction to fly to reacquire it. The yellow lights on the cross alternate randomly such that they illuminate one at a time, creating a faster, more demanding tracking task. The pilot must supply inputs at higher frequencies, which may uncover flaws in the simulated aircraft's handling qualities. The up-and-away task is designed to represent a real-world experiment, flying behind a test aircraft with lights on the top and bottom of the vertical tail, as well as the left and right tips of the horizontal stabilizer.

The simulator formulates a quantitative score based on the cumulative difference between the position of the target and the direction the aircraft's nose is pointed. Instead of modeling the up-and-away target at a three-dimensional world object, the position of the cross is driven by a commanded pitch angle and heading angle passed from the simulation. This eliminates the need for accurate speed and altitude control, while still providing for a challenging tracking task that will work for any aircraft. It also eliminates the need for three-dimensional vector operations to calculate the angular error between the aircraft's attitude and the commanded attitude. In this manner, the cross always maintains the same distance off the nose, and oscillates about the aircraft's current altitude.

Computer Code

This section contains the computer routines and include files that create the graphics for the real-time flight simulation. For more extensive explanations of specific GL function calls, refer to the Silicon Graphics Iris-4D Series manuals, Graphics Library Programming Guide and Graphics Library Reference Manual: C Edition. At the time of this writing, copies of these manuals are available in the Flight Simulation Lab, located in the AMDAF building on campus. For more information, including access to the source code for the entire flight simulation, contact Dr. Biezad at (805) 756-5126.

```

/*****/
/* aerodata.h */
/*****/
#ifndef _AERODATA_H
#define _AERODATA_H

void aerodata(double *Z, double *V, float *M, float *Vk);

#endif

/*****/
/* visuals.h */
/*****/
#ifndef _VISUALS_H
#define _VISUALS_H

#include <stdio.h>
#include <gl/gl.h>
#include <math.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <errno.h>

/* The color indices are all defined relative to the
   starting index: GRID */
#define GRASS (GRID+1)
#define MYRED (GRID+2)
#define SKY (GRID+3)
#define HUD_CLR (GRID+4)
#define ASPHALT (GRID+5)
#define purple1 (GRID+6)
#define purple3 (GRID+7)
#define grey0 (GRID+8)
#define skid (GRID+9)
#define poleclr (GRID+10)
#define yellow (GRID+11)

/* Define names for graphics objects */
#define GROUND_OBJ 1
#define RUNWAY 2
#define WATER_LINE 3
#define PTCH_LDR 4
#define HDG_SCL 5

```

```

#define STATIC_OBJ 6
#define MTN_RANGE 7
#define STRIPES 8
#define POLES 9
#define FPM 10
#define HANGAR 11
#define BLIMP 12
#define TARGET 13
#define LIGHT_1 14
#define LIGHT_2 15
#define LIGHT_3 16
#define LIGHT_4 17

/* Function prototypes */
void startgraphics_(int* win_orig_x, int* win_orig_y, int* width,
                   int* height, int* gridcol, double *glideslope);
void make_world_objects(double *glideslope);
void make_HUD_objects();
void drawframe_(double* X, double* Y, double* Z, double* psi, double* theta,
                double* phi, double *V, double* zdot, double* alpha,
                double* beta, double *qbar, int* AccuracyFlag,
                int* dio, float* del_psi, float* del_theta, int* light,
                float *control_defs);
void stopgraphics_(void);
void my_viewing_transform(double* x, double* y, double* z,
                          double* rot_psi, double* rot_theta, double* rot_phi);
void make_mountains();
void genstripe(Icoord x1,Icoord y1,Icoord x2,Icoord y2);
void update_vertical_scale(int x_origin, int y_origin, float value);
void update_horizontal_scale(int x_origin, int y_origin, float value);

#endif /* ifndef _VISUALS_H */

/*****/
/* visuals.c */
/*****/
#include "visuals.h"
#include "aerodata.h"
#define MIDDLE_X (DELTA_X/2)
#define MIDDLE_Y (DELTA_Y/2)
#define SCALE_LENGTH 20
#define GRID_SIZE 100000
#define max_int 500000
#define mtn_dist 200000
#define mtn_left 100000

/* Define bit masks to determine which switches are
   activated on the flight box */
#define S1 0x0001
#define S2 0x0002
#define S3 0x0004
#define S4 0x0008
#define S5 0x0010
#define S6 0x0020
#define S7 0x0040
#define S8 0x0080
#define S9 0x0100

```

```

#define S10 0x0200
#define S11 0x0400
#define S12 0x0800
#define S13 0x1000
#define S14 0x2000
#define S15 0x4000
#define S16 0x8000

int GRID;
int DELTA_X, DELTA_Y;

/* Define the origins for the control deflection indicators */
int control_defs_x_orig[] = {15, 25, 15, 10}; /* aileron, elevator, */
int control_defs_y_orig[] = {33, 10, 8, 10}; /* rudder, throttle */

float aspect_ratio;
Tag VECTOR;

/*****
/* startgraphics_ contains the calls to initialize the graphics */
/* environment, define the colors and create the graphics */
/* objects for instancing. */
*****/
void startgraphics_(int* win_orig_x, int* win_orig_y, int* win_width,
                   int* win_height, int* gridcol, double *glideslope)
{
    /* GRID is an index into the color table, where the custom
       definitions will begin. This is necessary since different
       machines have different amounts of color table space
       available, and the regions that are already in use change
       accordingly */
    GRID = *gridcol;

    /* the width and height of the graphics window is requested
       from the sim: */
    DELTA_X = *win_width;
    DELTA_Y = *win_height;

    /* the aspect ratio of the window is calculated for the
       call to perspective() */
    aspect_ratio = (float)DELTA_X/DELTA_Y;

    /* define where the graphics window will appear */
    prefposition(*win_orig_x, *win_orig_x + DELTA_X,
                *win_orig_y, *win_orig_y + DELTA_Y);

    /* open the graphics window, and give it a title */
    winopen("Flight scene");

    /* define the matrix mode */
    mmode(MVIEWING);

    /* enable double buffering */
    doublebuffer();

    /* set up color map mode */
    cmode();

```

```

/* apply the initialization calls to the
   graphics environment */
gconfig();

/* associate red, green, and blue values
   to the custom elements of the color table */
mapcolor (GRID, 170, 170, 170);
mapcolor (GRASS, 30, 105, 0);
mapcolor (MYRED, 255, 20, 20);
mapcolor (SKY, 10, 20, 160);
mapcolor (HUD CLR, 50, 255, 0);
mapcolor (ASPHALT, 100, 100, 100);
mapcolor (purple1, 150, 80, 64);
mapcolor (purple3, 75, 40, 32);
mapcolor (grey0, 224, 224, 224);
mapcolor (skid, 28, 28, 28);
mapcolor (poleclr, 10, 10, 10);
mapcolor (yellow, 255, 255, 0);

/* define a dotted line style for the negative
   rungs of the pitch ladder */
deflinestyle(1,0x6666);

/* define the world objects */
make_world_objects(glideslope);

/* define the HUD overlay objects */
make_HUD_objects();

/* define the mountain objects */
make_mountains();
}

/*****
/* make_world_objects defines the graphics objects that will */
/* be instantiated to create the outside scene. For example, */
/* the ground plane, hangars, runways, etc. */
*****/
void make_world_objects(double *glideslope)
{
    long point0[2], point1[2];
    int i, j, x, y;
    float x_orig, y_orig, del_x, del_y, del_h;
    float up, out, spacing;
    Coord xl, yl, zl, z2;

    float v[4][3] = { /* this is the ground */
        {-500000.0, -500000.0, 0.0},
        { 500000.0, -500000.0, 0.0},
        { 500000.0,  500000.0, 0.0},
        {-500000.0,  500000.0, 0.0}};

    float rnwy[4][3] = { /* runway */
        {-100.0, 0.0, 0.0},
        { 100.0, 0.0, 0.0},
        { 100.0, 8500.0, 0.0},
        {-100.0, 8500.0, 0.0}};

```

```

Scoord parray[][2]={ /* tire marks */
    5, 8100,
    -7, 8090,
    -35, 7700,
    -30, 7300,
    -4, 6200,
    1, 6150,
    30, 7230,
    36, 7780,
};

float hangar1[][3] = { /* hangar panel */
    {0.0, 0.0, 0.0},
    {350.0, 0.0, 0.0},
    {350.0, 0.0, -180.0},
    {175.0, 0.0, -220.0},
    {0.0, 0.0, -180.0}
};

float hangar2[][3] = { /* hangar panel */
    {0.0, -1026.0, 0.0 },
    {0.0, 0.0, 0.0 },
    {0.0, 0.0, -180.0 },
    {0.0, -1026.0, -180.0 }
};

float hangar3[][3] = { /* hangar panel */
    {0.0, -1026.0, -180.0 },
    {0.0, 0.0, -180.0 },
    {175.0, 0.0, -220.0 },
    {175.0, -1026.0, -220.0 }
};

float hangar4[][3] = { /* hangar panel */
    {175.0, -1026.0, -220.0 },
    {175.0, 0.0, -220.0 },
    {350.0, 0.0, -180.0 },
    {350.0, -1026.0, -180.0 }
};

float hangar5[][3] = { /* hangar panel */
    {350.0, -1026.0, -180.0 },
    {350.0, 0.0, -180.0 },
    {350.0, 0.0, 0.0 },
    {350.0, -1026.0, 0.0 }
};

float hangar6[][3] = { /* hangar panel */
    {0., -1026., 0. },
    {0., -1026., -180. },
    {175.0, -1026.0, -220.0 },
    {350.0, -1026.0, -180.0 },
    {350.0, -1026.0, 0.0 }
};

float blimp2[][3] = { /* blimp hangar panel */
    {0., 100., 0.},
    {0., 1100., 0.},
    {100., 1050., -200.},
    {100., 150., -200.}
};

```

```

};

float blimp3[][3] = { /* blimp hangar panel */
    {100., 150., -200.},
    {100., 1050., -200.},
    {200., 1050., -200.},
    {200., 150., -200.}
};

float blimp4[][3] = { /* blimp hangar panel */
    {200., 150., -200.},
    {200., 1050., -200.},
    {300., 1100., 0.},
    {300., 100., 0}
};

float blimp5[][3] = { /* blimp hangar panel */
    {100., 1200., 0.},
    {200., 1200., 0.},
    {200., 1050., -200.},
    {100., 1050., -200.}
};

float blimp6[][3] = { /* blimp hangar panel */
    {100., 1200., 0.},
    {100., 1050., -200.},
    {0., 1100., 0.}
};

float blimp7[][3] = { /* blimp hangar panel */
    {200., 1200., 0.},
    {300., 1100., 0.},
    {200., 1050., -200.}
};

/* define the GROUND_OBJ graphics object: the large green
square 'world,' with a grey grid for visual motion cue */
makeobj(GROUND_OBJ);
rotate(900,'x');
color(GRASS);
bgnpolygon();
    v3f(v[0]);
    v3f(v[1]);
    v3f(v[2]);
    v3f(v[3]);
endpolygon();
color(GRID);
point0[1] = -GRID_SIZE;
point1[1] = GRID_SIZE;
for (i = -GRID_SIZE; i <= GRID_SIZE; i+= 10000) {
    point0[0] = i;
    point1[0] = i;
    bgnline();
    v2i(point0);
    v2i(point1);
    endlne();
}
point0[0] = -GRID_SIZE;
point1[0] = GRID_SIZE;

```

```

    for (j = -GRID_SIZE; j <= GRID_SIZE; j+= 10000) {
        point0[1] = j;
        point1[1] = j;
        bgnline();
        v2i(point0);
        v2i(point1);
        endlne();
    }
closeobj();

/* define the RUNWAY graphics object: two grey rectangles
   that will be the runways */
makeobj(RUNWAY);
color(ASPHALT);
bgnpolygon();
    v3f(rnwy[0]);
    v3f(rnwy[1]);
    v3f(rnwy[2]);
    v3f(rnwy[3]);
endpolygon();
color(GRID);
rectfs(500, -800, 700, 8400);
closeobj();

/* define the STRIPES graphics object: all of the runway
   markings for 36L */
makeobj(STRIPES);
/* Tire marks */
color(skid);
pclf2s(8,parray);

color(GRID);
    for (y = 450; y < 8000; y+=350)          /* center stripes      */
        rectfs (-1,y,2,y+150);
    rectfs (72,0,75,8500);                    /* long side stripes      */
    rectfs (-72,0,-75,8500);
    for (x=8; x<=53; x+= 15)                  /* 4 big ones             */
        genstripe (x,10,x+10,150);
    for (x=38; x<=58; x+= 10)                 /* 3 small ones           */
        genstripe (x,500,x+5,575);
    genstripe (38,1000,63,1150);              /* 1 big fat one          */
    genstripe (38,1500,43,1575);
    genstripe (38,1500,43,1575);              /* 2                      */
    genstripe (48,1500,53,1575);
    genstripe (38,2000,43,2075);              /* 2                      */
    genstripe (48,2000,53,2075);
    genstripe (38,2500,43,2575);              /* 1                      */
    genstripe (38,3000,43,3075);              /* 1                      */
closeobj();

/* set up the geometry for the glideslope indicating
   "telephone poles;" convert all angles to radians */
out = 5.0/57.296; /* included angle between the two rows */
up = (*glideslope)/57.296; /* glideslope angle */
spacing = 250.0; /* distance between each pole */
x1 = x_orig = 0.0; /* x coordinate of the touchdown point */
y1 = y_orig = 8000.0; /* y coordinate of the touchdown point */
z1 = z2 = 0.0; /* z coordinate of the touchdown point */

```

```

/* deltas between each pole */
del_x = spacing*sinf(out);
del_y = spacing*cosf(out);
del_h = del_y*sinf(up);

/* increment x and y to start drawing poles away
   from the touchdown point */
x1 += del_x;
y1 += del_y;

/* create the POLES graphics object: the glideslope indicating
   "telephone poles" */
makeobj(POLES);
color(poleclr);
for (i = 0; i < 10; i++) {
    move(x1, y1, z1); /* move the graphics 'pen' */
    z2 -= del_h;      /* positive z is down */
    draw(x1, y1, z2); /* draw a pole */
    move(-x1, y1, z1); /* move to the other side */
    draw(-x1, y1, z2); /* draw the opposite pole */
    x1 += del_x;      /* increment x for next pole's base */
    y1 += del_y;      /* increment y for next pole's base */
}
closeobj();

/* create the HANGAR graphics object: a P-3 hangar that gets
   instanced twice */
makeobj(HANGAR);
color(skid);
rectfs(-100, 100, 450, -1126);
color(ASPHALT);
polf(5, hangar1);
color(grey0);
polf(4, hangar2);
polf(4, hangar4);
color(skid);
polf(4, hangar3);
closeobj();

/* create the BLIMP graphics object: the blimp hangar */
makeobj(BLIMP);
color(ASPHALT);
polf(4, blimp2);
color(skid);
polf(4, blimp3);
rectfs(-100, 1300, 400, -100);
color(grey0);
polf(4, blimp4);
color(poleclr);
polf(4, blimp5);
color(ASPHALT);
polf(3, blimp6);
polf(3, blimp7);
closeobj();
}

/*****
/* make_HUD_objects creates the graphics objects for the */

```



```

/* heads-up-display (HUD) overlay symbology */
/******/
void make_HUD_objects()
{
    char text_buffer[32];
    char *heading_num[] = {
        "33", "34", "35", "00", "01", "02", "03", "04", "05", "06",
        "07", "08", "09", "10", "11", "12", "13", "14", "15", "16",
        "17", "18", "19", "20", "21", "22", "23", "24", "25", "26",
        "27", "28", "29", "30", "31", "32", "33", "34", "35", "00",
        "01", "02", "03"};

    char *ladder_num[] = {
        "-90", "-80", "-70", "-60", "-50", "-40", "-30",
        "-20", "-10", "10", "20", "30", "40", "50", "60",
        "70", "80", "90"};

    float local_y = -162.5;
    int i, j, x, y;

    /* define the center of the window in window coordinates */
    x = 75;
    y = 50;

    /* define the FPM graphics object: the velocity vector */
    makeobj(FPM); /* Flight Path Marker (velocity vector) */
    pushmatrix();
    maketag(VECTOR);
    translate(0.0, 0.0, 0.0);
    move2s(x+2,y); draw2s(x+1,y); /* Draw Velocity Vector */
    draw2s(x,y+1); draw2s(x,y+2);
    move2s(x-2,y); draw2s(x-1,y);
    move2s(x,y+1); draw2s(x-1,y);
    draw2s(x,y-1); draw2s(x+1,y);
    popmatrix();
    closeobj();

    /* define the PTCH_LDR graphics object: the pitch ladder */
    makeobj(PTCH_LDR);
    j=0;

    /* make lines dotted for the negative pitch rungs */
    setlinestyle(1);

    /* draw and label the major negative rungs: */
    for (i= -9; i<0; i++){
        move2i(60, 52+i*25); /* draw one side of the rung */
        draw2i(60, 50+i*25);
        draw2i(70, 50+i*25);
        move2i(80, 50+i*25); /* draw the other side */
        draw2i(90, 50+i*25);
        draw2i(90, 52+i*25);
        cmov2(53, 50+i*25); /* label one side */
        charstr(ladder_num[j]);
        cmov2(92, 50+i*25); /* label the other side */
        charstr(ladder_num[j]);
        j++;
    }

    /* draw the negative minor rungs between the major ones */

```

```

for (i= 0; i<9; i++){
    move2(65, local_y);
    draw2(70, local_y);
    move2(80, local_y);
    draw2(85, local_y);
    local_y += 25;
};

/* draw the positive pitch rungs with solid lines */
setlinestyle(0);

/* draw the zero-rung, or horizon line */
move2i(50, 50);
draw2i(70, 50);
move2i(80, 50);
draw2i(100,50);

/* draw and label the positive major rungs */
for (i= 1; i<=9; i++){
    move2i(60, 48+i*25); /* draw left side */
    draw2i(60, 50+i*25);
    draw2i(70, 50+i*25);
    move2i(80, 50+i*25); /* draw right side */
    draw2i(90, 50+i*25);
    draw2i(90, 48+i*25);
    cmov2i(54, 48+i*25); /* label left side */
    charstr(ladder_num[j]);
    cmov2(92, 48+i*25); /* label right side */
    charstr(ladder_num[j]);
    j++;
}

/* draw positive minor rungs between the major ones */
for (i= 9; i<18; i++){
    move2(65, local_y);
    draw2(70, local_y);
    move2(80, local_y);
    draw2(85, local_y);
    local_y += 25;
}
closeobj();

/* define the HDG_SCL graphics object: the heading scale,
basically a horizontal tape strip that gets translated
left and right according to compass heading, and clipped
into a small box at the bottom of the window */
makeobj(HDG_SCL);
/* Major tics, every 10 degrees */
for (i=0; i<=420; i+=10){
    move2i(i,5);
    draw2i(i,8);
}
/* Minor tics, every 10 degrees, off-set from
major tics by 5 degrees */
for (i=5; i<=415; i+=10){
    move2i(i,5);
    draw2i(i,7);
}
/* Heading markers */
for (i=0; i<42; i-=1){

```

```

        cmov2(8.5+10*i,9);
        charstr(heading_num[i]);
    }
closeobj();

/* define the STATIC_OBJ graphics object: all of the static
stuff that only needs to be drawn once, such as the water
line, heading pointer, altitude and airspeed boxes, etc. */
makeobj(STATIC_OBJ);

/* Water Line */
move2i(75-4, 50);
draw2i(75-2, 50);
draw2i(75-1, 50-2);
draw2i(75, 50);
draw2i(75+1, 50-2);
draw2i(75+2, 50);
draw2i(75+4, 50);

/* Altitude and airspeed boxes */
rect(30, 60, 45, 65);
rect(105, 60, 120, 65);

/* Heading pointer */
move2i(74,3);
draw2i(76,3); draw2i(75,5); draw2i(74,3);

/* Put the text labels for the alpha and dynamic pressure */
/* digital readouts */
sprintf(text_buffer, "alpha:");
cmov2(3, 95);
charstr(text_buffer);

sprintf(text_buffer, "qbar:");
cmov2(3, 90);
charstr(text_buffer);

/* Draw static parts of control deflection indicators */
/* aileron */
move2i(control_defs_x_orig[0], control_defs_y_orig[0] + 1);
draw2i(control_defs_x_orig[0], control_defs_y_orig[0]);
draw2i(control_defs_x_orig[0]+SCALE_LENGTH, control_defs_y_orig[0]);
draw2i(control_defs_x_orig[0]+SCALE_LENGTH, control_defs_y_orig[0] + 1);
/* elevator */
move2i(control_defs_x_orig[1] -1, control_defs_y_orig[1]);
draw2i(control_defs_x_orig[1], control_defs_y_orig[1]);
draw2i(control_defs_x_orig[1], control_defs_y_orig[1]+SCALE_LENGTH);
draw2i(control_defs_x_orig[1] -1, control_defs_y_orig[1]+SCALE_LENGTH);
/* rudder */
move2i(control_defs_x_orig[2], control_defs_y_orig[2] +1);
draw2i(control_defs_x_orig[2], control_defs_y_orig[2]);
draw2i(control_defs_x_orig[2]+SCALE_LENGTH, control_defs_y_orig[2]);
draw2i(control_defs_x_orig[2]+SCALE_LENGTH, control_defs_y_orig[2] +1);
/* throttle */
move2i(control_defs_x_orig[3] -1, control_defs_y_orig[3]);
draw2i(control_defs_x_orig[3], control_defs_y_orig[3]);
draw2i(control_defs_x_orig[3], control_defs_y_orig[3]+SCALE_LENGTH);
draw2i(control_defs_x_orig[3] -1, control_defs_y_orig[3]+SCALE_LENGTH);

closeobj();

```

```

/* define the air-to-air cross TARGET object */
makeobj(TARGET);
    color(ASPHALT);
    rectf(-7.5, -1.25, 7.5, 1.25);
    rectf(-1.25, -7.5, 1.25, 7.5);
closeobj();

/* define the light on each arm of the cross as a
   separate graphics object to make it efficient to
   turn them on and off */
makeobj(LIGHT_1);
    color(yellow);
    rectf(-1.0, 5.25, 1.0, 7.25);
closeobj();

makeobj(LIGHT_2);
    color(yellow);
    rectf(5.25, -1.0, 7.25, 1.0);
closeobj();

makeobj(LIGHT_3);
    color(yellow);
    rectf(-1.0, -7.25, 1.0, -5.25);
closeobj();

makeobj(LIGHT_4);
    color(yellow);
    rectf(-7.25, -1.0, -5.25, 1.0);
closeobj();
}

/*****
/* drawframe_ is the routine that actually draws each graphics frame */
/* in the window and is called by the flight simulation after each */
/* round of several integration steps. The graphics transformation */
/* matrix is manipulated to put the outside objects in the correct */
/* perspective for the current aircraft position, then the world */
/* objects are instanced to render them to the frame. The HUD */
/* symbology is then manipulated and updated on the screen. */
*****/
void drawframe_(double* X, double* Y, double* Z, double* psi, double* theta,
                double* phi, double* V, double* zdot, double* alpha,
                double* beta, double* qbar, int* AccuracyFlag,
                int* dic, float* del_psi, float* del_theta, int* light,
                float* control_defs)
{
    static float local_psi, mach, vk;
    static float dx, dy;
    static char text_buffer[32];
    static int light_vec[] = {0, LIGHT_1, LIGHT_2, LIGHT_3, LIGHT_4};
    static float target_x, target_y;

    pushmatrix();
    perspective(400, aspect_ratio, 10.0, 400000.0);
    my_viewing_transform(X, Y, Z, psi, theta, phi);

    color(SKY);

```

```

clear();

/* Call world objects */
callobj(GROUND_OBJ);
callobj(MTN_RANGE);
translate(0.0, -20000.0, 0.0);

pushmatrix();
  translate(2100.0, 5200.0, 0.0);
  callobj(HANGAR);
  translate(-470.0, 0.0, 0.0);
  callobj(HANGAR);
  translate(-3000.0, -1200.0, 0.0);
  callobj(BLIMP);
popmatrix();

callobj(RUNWAY);
callobj(STRIPES);
if (S6 & *dio)
  callobj(POLES);

popmatrix();
ortho2(-0.5, 150.5, -0.5, 100.5);

/* Up and away task marker: */
if(*light) {
  pushmatrix();
    translate(75.0, 50.0, 0.0);
    rot(*phi, 'z');
    target_x = (*del_psi)*2.5;
    target_y = (*del_theta)*2.5;
    translate(target_x, target_y, 0.0);
    callobj(TARGET);
    callobj(light_vec[*light]);
    color(HUD_CLR);
    move2(0,0);
    draw2(-target_x, -target_y);
  popmatrix();
}

/* Call HUD objects */
color(HUD_CLR);
dx = (*beta)*2.5;
dy = -(*alpha)*2.5;
editobj(FPM);
  objreplace(VECTOR);
  translate(dx, dy, 0.0);
closeobj();
callobj(FPM);

pushviewport();
pushmatrix();
scrmask(.3*DELTA_X, .7*DELTA_X, .2*DELTA_Y, .8*DELTA_Y);
translate(75, 50, 0);
rot(*phi, 'z');
translate(0, -(*theta)*2.5, 0);
translate(-75, -50, 0);
callobj(PTCH_LDR);
popmatrix();
popviewport();

```

```

local_psi = (float) *psi;
while (local_psi < 0.0) local_psi += 360.0;

pushviewport();
pushmatrix();
scrmask(.4*DELTA_X, .6*DELTA_X, 0, .2*DELTA_Y);
translate(35.0-local_psi, 0, 0);
callobj(HDG_SCL);
popmatrix();
popviewport();

callobj(STATIC_OBJ);

/* Update digital displays */
aerodata(Z, V, &mach, &vk);

/* Altitude (ft) */
sprintf(text_buffer, "%.0f", -(Z));
cmov2(31, 61);
charstr(text_buffer);

/* Airspeed (knots true) */
sprintf(text_buffer, "%.0f", vk);
cmov2(106, 61);
charstr(text_buffer);

/* Vertical speed (ft/min) */
sprintf(text_buffer, "%.0f", -(zdot)*60.0);
cmov2(31, 57);
charstr(text_buffer);

/* Mach number */
sprintf(text_buffer, "%.2f", mach);
cmov2(106, 57);
charstr(text_buffer);

/* Draw pointers on control deflection indicators */
/* aileron */
update_horizontal_scale(control_defs_x_orig[0],
                        control_defs_y_orig[0],
                        control_defs[0]);

/* elevator */
update_vertical_scale(control_defs_x_orig[1],
                      control_defs_y_orig[1],
                      control_defs[1]);

/* rudder */
update_horizontal_scale(control_defs_x_orig[2],
                        control_defs_y_orig[2],
                        control_defs[2]);

/* commanded throttle */
update_vertical_scale(control_defs_x_orig[3],
                      control_defs_y_orig[3],
                      control_defs[3]);

/* show the actual thrust */
rect(control_defs_x_orig[3] - 1, control_defs_y_orig[3],
      control_defs_x_orig[3],
      control_defs_y_orig[3] + (int)(control_defs[4]*SCALE_LENGTH));

```

```

/* Digital displays for alpha and dynamic pressure;
   green if inside table-look-up envelope, red if outside */

   if (!(*AccuracyFlag)) color(MYRED); /* make follwing text red */

/* Alpha */
sprintf(text_buffer,"%1f", (*alpha));
cmov2(15,95);
charstr(text_buffer);

/* Dynamic Pressure */
sprintf(text_buffer,"%1f", (*qbar));
cmov2(15,90);
charstr(text_buffer);

swapbuffers();
}

/*****
/* stopgraphics_ closes the graphics gracefully */
*****/
void stopgraphics_()
{
    gexit();
}

/*****
/* my_viewing_transform manipulates the graphics transformation */
/* matrix through the proper operations so that it will translate */
/* and rotate the world objects into the proper perspective form */
/* the pilot 's point of view */
*****/
void my_viewing_transform(double* x, double* y, double* z,
                          double* rot_psi, double* rot_theta, double* rot_phi)
{
    Angle apsi, atheta, aphi;
    float sine, cosine;
    static Matrix mat = {1.0, 0.0, 0.0, 0.0,
                          0.0, 1.0, 0.0, 0.0,
                          0.0, 0.0, 1.0, 0.0,
                          0.0, 0.0, 0.0, 1.0};

    apsi = (Angle)(*rot_psi*10.0);
    atheta = (Angle)(*rot_theta*10.0);
    aphi = (Angle)(*rot_phi*10.0);

    gl_sincos(-aphi, &sine, &cosine);
    mat[0][0] = cosine;
    mat[0][1] = -sine;
    mat[1][0] = sine;
    mat[1][1] = cosine;
    multmatrix(mat);
    mat[0][0] = 1.0;
    mat[0][1] = 0.0;
    mat[1][0] = 0.0;

```

```

    gl_sincos(atheta, &sine, &cosine);
    mat[1][1] = cosine;
    mat[1][2] = -sine;
    mat[2][1] = sine;
    mat[2][2] = cosine;
    multmatrix(mat);
    mat[1][1] = 1.0;
    mat[1][2] = 0.0;
    mat[2][1] = 0.0;

    gl_sincos(apsi, &sine, &cosine);
    mat[0][0] = cosine;
    mat[0][2] = -sine;
    mat[2][0] = sine;
    mat[2][2] = cosine;
    multmatrix(mat);
    mat[0][2] = 0.0;
    mat[2][0] = 0.0;
    mat[2][2] = 1.0;

    translate(-(*y), *z, *x);
}

/*****
/* make_mountains defines the graphics objects to draw the simple */
/* silhouette mountains in the distance */
*****/
void make_mountains()
{
    register int x,y;
    Matrix mtn_range2;

    static long world[][3] = {
        {max_int,0,max_int},
        {-max_int,0,max_int},
        {-max_int,0,-max_int},
        {max_int,0,-max_int},
    };
    static long skyl[][3] = {
        {-max_int,0,-max_int},
        {-max_int,0,max_int},
        {0,max_int,0},
    };
    static long sky2[][3] = {
        {max_int,0,-max_int},
        {max_int,0,max_int},
        {0,max_int,0},
    };
    static long sky3[][3] = {
        {-max_int,0,-max_int},
        {max_int,0,-max_int},
        {0,max_int,0},
    };
    static long sky4[][3] = {
        {-max_int,0,max_int},
        {max_int,0,max_int},
    };
}

```



```

    {0,max_int,0},
};

static long north_mtn_lite1 [][][3] = {
    {-mtn_left-15000,0,-mtn_dist-5000},
    {-100000,8500,-mtn_dist-5000},
    {-80000,0,-mtn_dist-5000},
};

static long north_mtn_lite2 [][][3] = {
    {-90000,0,-mtn_dist-5000},
    {-70000,12000,-mtn_dist-5000},
    {-50000,5000,-mtn_dist-5000},
};

static long north_mtn_lite3 [][][3] = {
    {-70000,5000,-mtn_dist-5000},
    {-40000,21000,-mtn_dist-5000},
    {-20000,5000,-mtn_dist-5000},
};

static long north_mtn_lite4 [][][3] = {
    {-30000,5000,-mtn_dist-5000},
    {0,22500,-mtn_dist-5000},
    {50000,5000,-mtn_dist-5000},
};

static long north_mtn_lite5 [][][3] = {
    {12000,10000,-mtn_dist-5000},
    {28000,19000,-mtn_dist-5000},
    {37300,17500,-mtn_dist-5000},
};

static long north_mtn_lite6 [][][3] = {
    {12000,10000,-mtn_dist-5000},
    {50000,22000,-mtn_dist-5000},
    {80000,5000,-mtn_dist-5000},
};

static long north_mtn_dark0 [][][3] = {
    {-mtn_left-20000,0,-mtn_dist},
    {-mtn_left,6000,-mtn_dist},
    {mtn_left,6000,-mtn_dist},
    {mtn_left+20000,0,-mtn_dist},
};

static long north_mtn_dark1 [][][3] = {
    {-100000,5000,-mtn_dist},
    {-85000,10000,-mtn_dist},
    {-70000,5000,-mtn_dist},
};

static long north_mtn_dark2 [][][3] = {
    {-60000,5000,-mtn_dist},
    {-25000,20000,-mtn_dist},
    {0,5000,-mtn_dist},
};

```

```

static long north_mtn_dark3 [][][3] = {
    {-10000,5000,-mtn_dist},
    {5000,17500,-mtn_dist},
    {30000,5000,-mtn_dist},
};

static long north_mtn_dark4 [][][3] = {
    {15000,5000,-mtn_dist},
    {30000,15000,-mtn_dist},
    {50000,10000,-mtn_dist},
};

static long north_mtn_dark5 [][][3] = {
    {15000,5000,-mtn_dist},
    {62000,20000,-mtn_dist},
    {100000,5000,-mtn_dist},
};

makeobj (MTN_RANGE);
pushmatrix();
rotate(-900, 'x');
color (purple1);
polfi (3,north_mtn_lite1);
polfi (3,north_mtn_lite2);
polfi (3,north_mtn_lite3);
polfi (3,north_mtn_lite4);
polfi (3,north_mtn_lite5);
polfi (3,north_mtn_lite6);

color (purple3);
polfi (4,north_mtn_dark0);
polfi (3,north_mtn_dark1);
polfi (3,north_mtn_dark2);
polfi (3,north_mtn_dark3);
polfi (3,north_mtn_dark4);
polfi (3,north_mtn_dark5);

popmatrix();
closeobj ();

}

/*****/
/* Due to the two axes of symmetry for runway markings, genstripe */
/* will mirror each rectangle to make a total of four stripes for */
/* each one passed to it */
/*****/
void genstripe (Icoord x1,Icoord y1,Icoord x2,Icoord y2)
{
    rectfs (x1,y1,x2,y2);
    rectfs (-x1,y1,-x2,y2);
    rectfs (x1,8500-y1,x2,8500-y2);
    rectfs (-x1,8500-y1,-x2,8500-y2);

    return;
}

```

```

/*****
/* update_vertical_scale is used to draw the pointer on the */
/* commanded thrust and elevator position indicators      */
/*****
void update_vertical_scale(int x_origin, int y_origin, float value)
{
    int y_val;

    y_val = y_origin + (int)(value*SCALE_LENGTH);

    move2i(x_origin, y_val);
    draw2i(x_origin + 1, y_val + 1);
    draw2i(x_origin + 1, y_val - 1);
    draw2i(x_origin, y_val);

    return;
}

/*****
/* update_horizontal_scale is used to draw the pointer on the */
/* rudder and elevator position indicators                    */
/*****
void update_horizontal_scale(int x_origin, int y_origin, float value)
{
    int x_val;

    x_val = x_origin + (int)(value*SCALE_LENGTH);

    move2i(x_val, y_origin);
    draw2i(x_val + 1, y_origin - 1);
    draw2i(x_val - 1, y_origin - 1);
    draw2i(x_val, y_origin);

    return;
}

```